Assignment 6: A/D-Converters and PID Controllers

1DT056: Programming Embedded Systems
Uppsala University

March 4th, 2012

You can achieve a maximum number of 20 points in this assignment. 12 out of the 20 points are required to pass the assignment. Assignments are to be solved by students individually.

Since the week March 12-16 is an exam week (at least for some students), you have time until **March 19th** to submit solutions for the assignment. Submission details are given in the end of the assignment.

In this exercise, we will go through some examples of using analogue-to-digital converters (ADCs) of an ARM CORTEX M3 controller. ADCs are provided by most micro-controllers and convert, as the name tells, analogue (voltage) values to digital numbers. Common applications are the reading of sensor or audio data, converting to a format suitable for processing by software. ARM CORTEX M3 controllers typically contain a number of 12-bit ADCs (i.e., the produced digital values are 12 bit wide), each of which can read from a number of channels represented by pins of the controller.

Exercise 1 Simple conversions

The ADC of our micro-controller has to be initialised before data can be read from it. Starting from the same MDK-ARM project (`http://www.it.uu.se/edu/course/homepage/pins/vt12/lab_env.zip`) as before, three things have to be changed to this end:

- The source file `STM32F10xFWLib/src/stm32f10x_adc.c`, which provides firmware functions to access ADCs, has to be added to the group `Target 1/System`; this is done by right-clicking on the `System` group in the `Project` pane on the left and selecting this file.
- In the file `stm32f10x_conf.h`, the lines

  ```
  //#define _ADC
  //#define _ADC1
  ```

  have to be un-commented, to enable ADC firmware support.

- The following piece of initialisation code has to be put in the beginning of the main function in `main.c` (also available in `http://www.it.uu.se/edu/course/homepage/pins/vt12/adc_init_code.c`):

```
#include "stm32f10x_it.h"
#include "stm32f10x_adc.h"

int main( void )
{
  ADC_InitTypeDef ADC_InitStructure;

  prvSetupHardware();    // as before

  /* ADC clock: ADCCLK = PCLK2/4, here 18 MHz */
  RCC_ADCCLKConfig(RCC_PCLK2_Div4);
  RCC_APB2PeriphClockCmd( RCC_APB2Periph_ADC1, ENABLE );

  /* ADC1 configuration ------------------------------------*/

  // no dual ADC
  ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;
  // read from the channel(s) configured below
  ADC_InitStructure.ADC_ScanConvMode = ENABLE;
  // one-shot conversion
  ADC_InitStructure.ADC_ContinuousConvMode = DISABLE;
  // we only trigger conversion internally
  ADC_InitStructure.ADC_ExternalTrigConv =
    ADC_ExternalTrigConv_None;
  // store result in least significant bits
  ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;

  // only read from one channel at a time
  ADC_InitStructure.ADC_NbrOfChannel = 1;
  ADC_Init(ADC1, &ADC_InitStructure);

  /* Power up the ADC */
  ADC_Cmd(ADC1, ENABLE);

  /* Enable ADC1 reset calibaration register */
  ADC_ResetCalibration(ADC1);
  /* Check the end of ADC1 reset calibration register */
  while(ADC_GetResetCalibrationStatus(ADC1));

  /* Start ADC1 calibaration */
  ADC_StartCalibration(ADC1);
  /* Check the end of ADC1 calibration */
  while(ADC_GetCalibrationStatus(ADC1));

  // ...
}
```

After this initialisation, we can, in the simplest case, read the current voltage value on some particular ADC channel using statements like the following:

```
// Specify the channel to convert from (enough to do this once)
ADC_RegularChannelConfig(
  ADC1,                        // use ADC 1
  ADC_Channel_0,               // channel 0
  1,                           // rank (not relevant here)
```

```
  ADC_SampleTime_239Cycles5); // sample for 239.5 + 12.5 cycles
                              // (14 microseconds)

// Clear end-of-conversion flag
ADC_ClearFlag(ADC1, ADC_FLAG_EOC);
// Start the conversion
ADC_SoftwareStartConvCmd(ADC1, ENABLE);
// Wait until the result is available
while (!ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC));

printf("Value: %d\n", ADC_GetConversionValue(ADC1));
```

When using the μVision simulator/debugger, you can specify the analogue value to be converted as a number between 0.0 (corresponding to the digital value 0) and 3.3 (corresponding to the value 4095) in the dialogue "Peripherals → A/D Converters → ADC1", in the field `ADC1_IN0`.

To get started, re-code some of the examples that you implemented in Lustre in the previous Assignment 5 (`http://www.it.uu.se/edu/course/homepage/pins/vt12/assignment5.pdf`) in C. Your programs should contain a task that periodically (with a period of 100ms) reads inputs from the ADC and GPIO pins, and generates outputs that are visualised using the `printf` function.

1. Implement a C program corresponding to the `SumReset` Lustre program you wrote in Assignment 5, Exercise 2.1. Read the input `X` (as an integer value between 0 and 4095) from ADC 1, channel 0, and the input `Reset` from GPIO C, Pin 0 (like in Assignment 2, Exercise 1). The output of the program is supposed to look like this: **(3p)**

   ```
   S=0
   S=42
   S=64
   S=0
   ...
   ```

2. Give a similar implementation for the `Average` Lustre program from Assignment 5, Exercise 2.2. **(3p)**
3. Give a similar implementation for the `HasHappenedWithin` Lustre program from Assignment 5, Exercise 2.3. **(3p)**

Exercise 2   PID cruise controller

This exercise is a simple case study of a closed-loop controller, a cruise control system that manipulates the throttle of a car engine in order to establish and maintain some desired speed. Our cruise control system has two (real-valued) inputs:

- The measured actual speed v_a of the vehicle.
- The desired speed v_t chosen by the driver (the *setpoint*).

The system has one (real-valued) output:

- The value p of the throttle of the car engine, controlling the force that the engine applies to accelerate the vehicle. We assume that $p = 0$ corresponds to closed throttle (no force), and $p = 1$ corresponds to full throttle.

The goal of the cruise control system is to establish the equation $v_a = v_t$ by controlling p over time. If $v_t > v_a$, the control system can open up throttle to accelerate the car; if $v_t < v_a$, the throttle can be closed, which means that friction and air resistance will slow down the car.

In order to implement such a cruise control system using an ARM CORTEX M3 micro-controller, we assume the following mapping of inputs and outputs to micro-controller ports:

v_a	A/D Converter 1, Channel 0
v_t	A/D Converter 1, Channel 1
p	Timer 3, Channel 1 (using pulse-width modulation)

In case of v_a, v_t, we assume that the analogue input value 3.3 (corresponding to the digital value 4095) represents the speed $40\,m/s$.

We provide a skeleton project for the cruise control system in the following archive: `http://www.it.uu.se/edu/course/homepage/pins/vt12/cruise_skeleton.zip`. The file `main.c` already contains code for initialising the ADC and timer.

To implement the actual cruise control algorithm, use a *proportional-integral-derivative (PID) controller,* which is the most common kind of controller and implements (approximates) the following equation:

$$p(t) \;=\; K_p e(t) + K_i \int_0^t e(s)ds + K_d\, \dot{e}(t) \qquad (1)$$

where:

- $p(t)$ is the output (throttle) generated by the controller at time t;
- $e(t)$ is the error between the measured value of the process variable (the actual speed of the vehicle) and the setpoint (the desired speed); that means, $e(t) = v_t(t) - v_a(t)$;
- K_p is the coefficient of the *proportional term (gain),* which creates a direct influence of the measured error on the output;
- K_i is the coefficient of the *integral term,* which sums up the error over time and is useful to amplify and eliminate small, persistent errors;
- K_d is the coefficient of the *derivative term,* which makes the change rate of the error over time contribute to the controller output, and can be used to slow down the change rate and to prevent overshooting.

```
oldError ← 0;
sum ← 0;
while true do
    read inputs v_a, v_t;
    error ← v_t − v_a;
    sum ← sum + T · error;
    generate output p = K_p error + K_i sum + K_d (error−oldError)/T;
    oldError ← error;
    delay for time T;
end
```

Algorithm 1: PID-controller

K_p, K_i, K_d are parameters that have to be chosen when tuning the controller for some particular application (also see the next question).

1. Implement the actual control algorithm by adding a time-triggered task to the system that periodically reads the inputs v_a and v_t and updates the output p. To this end, (1) has to be discretised, which in the simplest case (with sampling period T) results in: **(6p)**

$$p(n) \;=\; K_p\, e(n) + K_i\, T \sum_{i=0}^{n} e(i) + K_d\, \frac{e(n) - e(n-1)}{T}$$

   In pseudo-code, this looks as shown in Algorithm 1.

2. Tune the parameters K_p, K_i, K_d to establish a well-behaved system. This is often done by first setting $K_i = K_d = 0$, in order to determine good coefficients K_p for the proportional term alone. Modify K_i, K_d to improve the controller behaviour only after you have learnt about reasonable values for K_p. **(5p)**
   To test the controller, use the μVision debug function provided in the file `simulator.ini` (included in `http://www.it.uu.se/edu/course/homepage/pins/vt12/cruise_skeleton.zip`), which simulates the behaviour of the car, as well as external influences such as friction and wind. A run of this script will produces output similar to the lower part of Fig. 1. It can also be interesting to plot the evolution of the values v_a, v_t, p using the "Logic Analyzer," which produces graphs like the upper part of Fig. 1.
   Aspects that should be optimised are:
   - the time needed to reach a stable state after changed the setpoint, or after occurrence of external factors such as wind.
   - the precision of the controller once a stable state has been reached ("steady-state error").
   - potential overshooting that occurs after changing the setpoint.

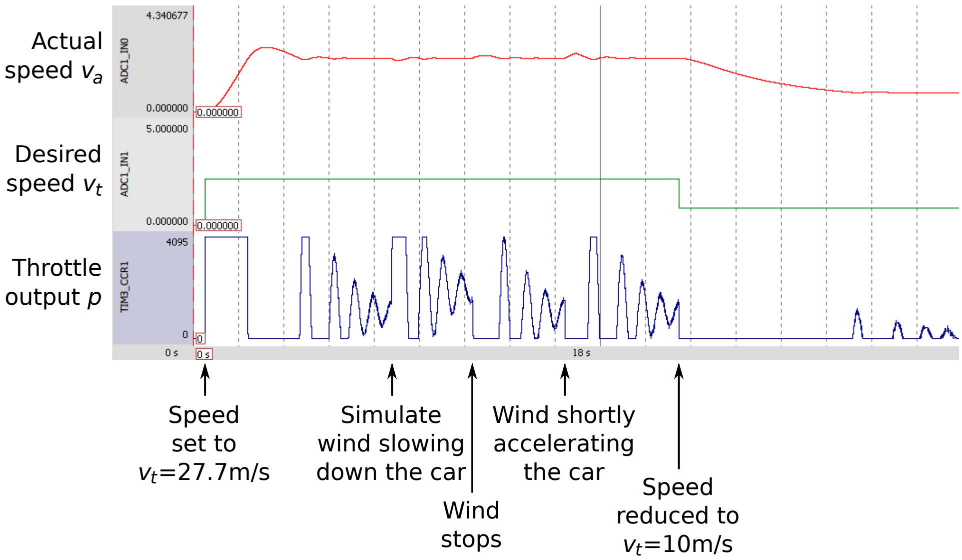
The first two criteria are measured and printed by `simulator.ini` (the lines starting with `1.`, `2.`, etc.). Report your experiences and give the best parameters that you could find (which hopefully give better results than shown in Fig. 1).

A useful optimisation that you might want to implement is *anti-windup:* since the throttle output p is bounded, it can happen that the value of the *sum* variable grows very large when the setpoint v_t is suddenly changed by a larger value. A possible solution to this is to disable the integral term (and to freeze *sum*) at times when p reaches the upper or lower limit.

Submission

Solutions to this assignment are to be submitted
via email to othmane.rezine@it.uu.se before the lecture/exercise on
Monday, March 19th, 2012, 13:15 (room 1245).

Make sure that your solution is clearly marked with your name and
personal number. No solutions will be accepted after the exercise.




```
Setting target speed to 27.7m/s
1. Stable after 5.230000s, average speed deviation 0.118110m/s
Setting wind to 10.0m/s
2. Stable after 0.610000s, average speed deviation 0.155034m/s
Setting wind to 0.0m/s
3. Stable after 1.055000s, average speed deviation 0.135379m/s
Setting wind to -20.0m/s
Setting wind to 0.0m/s
4. Stable after 1.565000s, average speed deviation 0.113251m/s
Setting target speed to 10.0m/s
5. Stable after 7.225000s, average speed deviation 0.119950m/s
```

Figure 1: Output of the script `simulator.ini`, together with plots showing a possible evolution of the values v_a, v_t, p over time